ONDotnet.com: .NET **Serialization**, Part 1 [Oct. 13, 2003]
... serialization. In the next article, I will discuss how to **serialize** an **object** into an **XML** document. Binary **Serialization**. Consider ...
www.oreillynet.com/pub/a/dotnet/2003/ 10/13/serializationpt1.html - 36k - Mar 1, 2004 - Cached - Similar pages

Serializing an Immutable Bean Property to **XML** (Java Developers ...
... Create an **object** with an ... new FileOutputStream("outfilename.**xml**")); Specify to ... class, new DefaultPersistenceDelegate(propertyNames); // **Serialize** the **object** ...
javaalmanac.com/egs/java.beans/Immute.html - 6k - Cached - Similar pages

Gooooooooogle ▶
Result Page: 1 2 3 4 5 6 7 8 9 10 **Next**

Google

Advanced Search Preferences Language Tools Search Tips

serialize and xml and object and Google Search

The "AND" operator is unnecessary -- we include all search terms by default. [details]

Web · Images · Groups · Directory · News ·

Searched the web for serialize and xml and object and persist. Results 1 - 10 of about 6,460. Search took 0.37

Fawcette.com - **Serialize Objects** to **XML** in .NET
... Let's look at a simple example of using **serialization** to **persist** an **object's** state to **XML**. This code shows a simple class and associated ...
www.fawcette.com/xmlmag/2002_05/ online/xml_dwahlin_05_20_02/ - 27k - Mar 1, 2004 - Cached - Similar pages

Visual Studio Magazine - Build **Serialized Objects**
... Using the **objects** contained in the System.**XML**.**Serialization** namespace, you can create a **serialized** configuration **object** that you can **persist** and de-**persist** to ...
www.fawcette.com/vsm/2002_01/ magazine/features/jgoodyear/ - 31k - Cached - Similar pages
[More results from www.fawcette.com]

XML.com: **XML** & **Object** Persistence: **Serialization** Problems [Sep. 08 ...
... of **serialized** data) is taken care of by the **XML** parser ... So far we've seen how to **serialize** simple data types ... at Figure 1 you see not just one **object**, but several ...
www.xml.com/pub/a/1999/09/serialization/index2.html - 36k - Cached - Similar pages

Serialize an **XML** DOM **Object** to a File (Visual Basic) (MSXML 4.0 ...
Serialize an **XML** DOM **Object** to a File (Visual Basic). You might often need to **persist** a DOM **object** so that you can reuse it later, or to save the **XML object** ...
msdn.microsoft.com/library/en-us/ xmlsdk/htm/dom_hdi_vb_2xyd.asp - 12k - Cached - Similar pages

Walkthrough: Persisting an **Object** in Visual Basic .NET (Visual ...
... To **persist** the **object** using SOAP format: In ... module: Imports System.Runtime.**Serialization**.Formatters.Soap; ... references from "SavedLoan.bin" to "SavedLoan.**xml**" . ...
msdn.microsoft.com/library/en-us/vbcon/html/ vbwlkwalkthroughpersistingobject.asp - 22k - Cached - Similar pages
[More results from msdn.microsoft.com]

Guides
... If you use binary **serialization** to **persist** an **object**, deserializing it will give you an exact copy. **XML serialization**, on the other hand, will only **persist** ...
www.informit.com/ guides/content.asp?g=dotnet&seqNum=157 - 30k - Mar 1, 2004 - Cached - Similar pages

TopXML : **Serializing** an **object** (.NET Framework)
... The. **Serialize**() method makes use of the pluggable architecture we ... from System.IO.Stream, System.**Xml**.XmlWriter or ... deal of flexibility where to **persist objects** to ...
www.topxml.com/xmlserializer/serializing_an_object.asp - 25k - Mar 1, 2004 - Cached - Similar pages

TopXML : Advanced XmlSerializer (.NET Framework)
... Using **XML serialization** will reduce the amount of code you have ... You no longer have to parse **XML** to initialize ... you have to develop code for **objects** to **persist** ...
www.topxml.com/xmlserializer/ advanced_xmlserializer.asp - 24k - Mar 1, 2004 - Cached - Similar pages
[More results from www.topxml.com]

Serializing Objects

Reconstructing an object from a stream requires that the object first be written to a stream. So let's start there.

How to Write to an ObjectOutputStream

Writing objects to a stream is a straightforward process. For example, the following gets the current time in milliseconds by constructing a `Date` object and then serializes that object:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

`ObjectOutputStream` must be constructed on another stream. This code constructs an `ObjectOutputStream` on a `FileOutputStream`, thereby serializing the object to a file named `theTime`. Next, the string `Today` and a `Date` object are written to the stream with the `writeObject` method of `ObjectOutputStream`.

Thus, the `writeObject` method serializes the specified object, traverses its references to other objects recursively, and writes them all. In this way, relationships between objects are maintained.

`ObjectOutputStream` implements the `DataOutput` interface that defines many methods for writing primitive data types, such as `writeInt`, `writeFloat`, or `writeUTF`. You can use these methods to write primitive data types to an `ObjectOutputStream`.

The `writeObject` method throws a `NotSerializableException` if it's given an object that is not serializable. An object is serializable only if its class implements the `Serializable` interface.

How to Read from an ObjectInputStream

Once you've written objects and primitive data types to a stream, you'll likely want to read them out again and reconstruct the objects. This is also straightforward. Here's code that reads in the `String` and the `Date` objects that were written to the file named `theTime` in the previous example:

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Like `ObjectOutputStream` , `ObjectInputStream` must be constructed on another stream. In this example, the objects were archived in a file, so the code constructs an `ObjectInputStream` on a `FileInputStream` . Next, the code uses `ObjectInputStream` 's `readObject` method to read the `String` and the `Date` objects from the file. The objects must be read from the stream in the same order in which they were written. Note that the return value from `readObject` is an object that is cast to and assigned to a specific type.

The `readObject` method deserializes the next object in the stream and traverses its references to other objects recursively to deserialize all objects that are reachable from it. In this way, it maintains the relationships between the objects.

`ObjectInputStream` stream implements the `DataInput` interface that defines methods for reading primitive data types. The methods in `DataInput` parallel those defined in `DataOutput` for writing primitive data types. They include methods such as `readInt` , `readFloat` , and `readUTF` . Use these methods to read primitive data types from an `ObjectInputStream` .

Start of Tutorial > Start of Trail > Start of Lesson Search
Feedback Form

Object Serialization

Two streams in `java.io` -- `ObjectInputStream` and `ObjectOutputStream` -- are run-of-the-mill byte streams and work like the other input and output streams. However, they are special in that they can read and write objects.

The key to writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read. Thus reading and writing objects is a process called *object serialization*. Object serialization is essential to building all but the most transient applications. You can use object serialization in the following ways:

- Remote Method Invocation (RMI)--communication between objects via sockets

  Note: The client and server programs in Putting It All Together use RMI to communicate. You can see object serialization used in that example to pass various objects back and forth between the client and server.

- Lightweight persistence--the archival of an object for use in a later invocation of the same program.

You need to know about object serialization from two points of view. First, you need to know how to serialize objects by writing them to an `ObjectOutputStream` and reading them in again using an `ObjectInputStream` . The next section, Serializing Objects, shows you how. Second, you will want to know how to write a class so that its instances can be serialized. You can read how to do this in the section after that, Providing Object Serialization for Your Classes.

Providing Object Serialization for Your Classes

An object is serializable only if its class implements the `Serializable` interface. Thus, if you want to serialize the instances of one of your classes, the class must implement the `Serializable` interface. The good news is that `Serializable` is an empty interface. That is, it doesn't contain any method declarations; its purpose is simply to identify classes whose objects are serializable.

Implementing the Serializable Interface

Here's the complete definition of the `Serializable` interface:

```
package java.io;
public interface Serializable {
    // there's nothing in here!
};
```

Making instances of your classes serializable is easy. You just add the `implements Serializable` clause to your class declaration like this:

```
public class MySerializableClass implements Serializable {
    ...
}
```

You don't have to write any methods. The serialization of instances of this class are handled by the `defaultWriteObject` method of `ObjectOutputStream`. This method automatically writes out everything required to reconstruct an instance of the class, including the following:

- Class of the object
- Class signature
- Values of all non-`transient` and non-`static` members, including members that refer to other objects

You can deserialize any instance of the class with the `defaultReadObject` method in `ObjectInputStream`.

For many classes, this default behavior is good enough. However, default serialization can be slow, and a class might want more explicit control over the serialization.

Customizing Serialization

You can customize serialization for your classes by providing two methods for it: `writeObject` and `readObject`. The `writeObject` method controls what information is saved and is typically used to append additional information to the stream. The `readObject` method either reads the information written by the corresponding `writeObject` method or can be used

to update the state of the object after it has been restored.

The `writeObject` method must be declared exactly as shown in the following example and should call the stream's `defaultWriteObject` as the first thing it does to perform default serialization. Any special arrangements can be handled afterwards:

```
private void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
    // customized serialization code
}
```

The `readObject` method must read in everything written by `writeObject` in the same order in which it was written. Also, the `readObject` method can perform calculations or update the state of the object. Here's the `readObject` method that corresponds to the `writeObject` method just shown:

```
private void readObject(ObjectInputStream s) throws IOException  {
    s.defaultReadObject();
    // customized deserialization code
    ...
    // followed by code to update the object, if necessary
}
```

The `readObject` method must be declared exactly as shown.

The `writeObject` and `readObject` methods are responsible for serializing only the immediate class. Any serialization required by the superclasses is handled automatically. However, a class that needs to explicitly coordinate with its superclasses to serialize itself can do so by implementing the `Externalizable` interface.

Implementing the Externalizable Interface

For complete, explicit control of the serialization process, a class must implement the `Externalizable` interface. For `Externalizable` objects, only the identity of the object's class is automatically saved by the stream. The class is responsible for writing and reading its contents, and it must coordinate with its superclasses to do so.

Here's the complete definition of the `Externalizable` interface that extends `Serializable`:

```
package java.io;
public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out) throws IOException;
    public void readExternal(ObjectInput in) throws IOException,
                             java.lang.ClassNotFoundException;
}
```

The following holds for an `Externalizable` class:

- It must implement the `java.io.Externalizable` interface.
- It must implement a `writeExternal` method to save the state of the object. Also, it must explicitly coordinate with its supertype to save its state.
- It must implement a `readExternal` method to read the data written by the `writeExternal` method from the stream and restore the state of the object. It must explicitly coordinate with the supertype to restore its state.

- If externally defined format is being written, the `writeExternal` and `readExternal` methods are solely responsible for that format.

The `writeExternal` and `readExternal` methods are public and carry the risk that a client may be able to write or read information in the object other than by using its methods and fields. These methods must be used only when the information held by the object is not sensitive or when exposing that information would not present a security risk.

Protecting Sensitive Information

When developing a class that provides controlled access to resources, you must take care to protect sensitive information and functions. During deserialization, the private state of the object is restored. For example, a file descriptor contains a handle that provides access to an operating system resource. Being able to forge a file descriptor would allow some forms of illegal access, since restoring state is done from a stream. Therefore the serializing runtime must take the conservative approach and not trust the stream to contain only valid representations of objects. To avoid compromising a class, you must provide either that the sensitive state of an object must not be restored from the stream or that it must be reverified by the class.

Several techniques are available to protect sensitive data in classes. The easiest is to mark fields that contain sensitive data as private `transient`. `transient` and `static` fields are not serialized or deserialized. Marking the field will prevent the state from appearing in the stream and from being restored during deserialization. Since writing and reading (of private fields) cannot be superseded outside of the class, the class's `transient` fields are safe.

Particularly sensitive classes should not be serialized. To accomplish this, the object should not implement either the `Serializable` or `Externalizable` interface.

Some classes may find it beneficial to allow writing and reading but to specifically handle and revalidate the state as it is deserialized. The class should implement `writeObject` and `readObject` methods to save and restore only the appropriate state. If access should be denied, throwing a `NotSerializableException` will prevent further access.